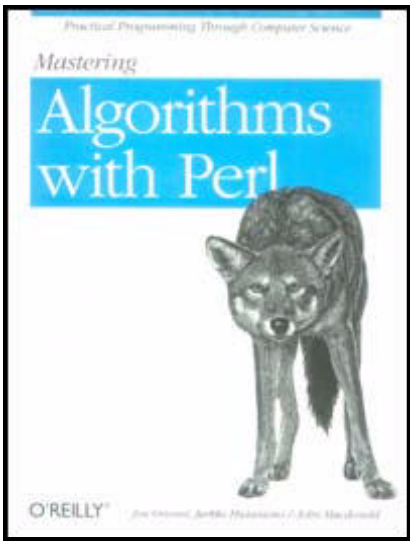
EXHIBIT 14



Mastering Algorithms with Perl

Jon Orwant, Jarkko Hietaniemi,
and John Macdonald

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Mastering Algorithms with Perl
by Jon Orwant, Jarkko Hietaniemi. and John Macdonald

Printed in the United States of America.
Cover illustration by Lorrie LeJeune, Copyright © 1999 O'Reilly & Associates, Inc.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editors: Andy Oram and Jon Orwant

Production Editor: Melanie Wang

Printing History:

August 1999: First Edition.

ISBN: 1-56592-398-7 [1/00]
[M]]break

Table of Contents

```
            && $self->{board}[$self->{next_move}] ne $empty;


        $self->{next_move} = undef if $self->{next_move} == 9;
        return $self->{next_move};
    }


    # Create the new position that results from making a move.
    sub make_move {
        my $self = shift;
        my $move = shift;


        # Copy the current board, changing only the square for the move.
        my $myturn = $self->{turn};
        my $newboard = [ @{$self->{board}} ];
        $newboard->[$move] = $move[$myturn];
```



```
    return tic_tac_toe->new(1 - $myturn, $newboard);
}


# Get the cached evaluation of this position.
sub evaluate {
    my $self = shift;


    return $self->{score};
}


# Display the position.
sub description {
    my $self = shift;
    my $board = $self->{board};
    my $desc = "@$board[0..2]\n@$board[3..5]\n@$board[6..8]\n";
    return $desc;
}


sub best_rating {
    return 101;
}
```

Exhaustive Search

The technique of generating and analyzing all of the possible states of a situation is called *exhaustive search*. An exhaustive search is the generative analog of linear search—try everything until you succeed or run out of things to try. (Exhaustive search has also been called the British Museum Search, based on the light-hearted idea that the only way to find the most interesting object in the British Museum is to plod through the entire museum and examine everything. If your data structure, like the British Museum, does not order its elements according to how interesting they are, this technique may be your only hope.)

Consider a program that plays chess. If you were determined to use a lookup search, you might want to start by generating a data structure containing all possible chess positions. Positions could be linked wherever a legal move leads from one position to another. Then, identify all of the final positions as "win for white," "win for black," or "tie," labeling them W, B, and T, respectively. In addition, when a link leads to a labeled position, label the link with the same letter as the position it leads to.

Next, you'd work backwards from identified positions. If a W move is available from a position where it is white's turn to move, label that position W too (and remember the move that leads to the win). That determination can be made regardless of whether the other moves from that position have been identified yet—white can choose to win rather than move into unknown territory. (A similar check finds positions where it is black's move and a B move is available.) If there is no winning move available, a position can only be identified if *all* of the possi-soft

ble moves have been labeled. In such a case, if any of the available moves is T, so is the position; but if all of the possible moves are losers, so is the position (i.e., B if it is white's turn, or W if it is black's turn). Repeat until all positions have been labeled.

Now you can write a program to play chess with a lookup search—simply lookup the current position in this data structure, and make the preferred move recorded there, an $O(1)$ operation. Congratulations You have just solved chess. White's opening move will be labeled W, T, or B. Quick, publish your answer—no one has determined yet whether white has a guaranteed win (although it would come as quite a shock if you discovered that black does).

There are a number of problems, however. Obviously, we skipped a lot of detail—you'd need to use a number of algorithms from Chapter 8, *Graphs*, to manage the board positions and the moves between them. We've glossed over the possibilities of draws that occur because of repeated positions—more graph algorithms to find loops so that we can check them to see whether either player would ever choose to leave the loop (because he or she would have a winning position).

But the worst problem is that there are a lot of positions. For white's first move, there are 20 different possibilities. Similarly, for black's first move. After that, the number of possible moves varies—as major pieces are exposed, more moves become available, but as pieces are captured, the number decreases.

A rough estimate says that there are about 20 choices for each possible turn, and a typical game lasts about 50 moves, which gives 20^{50} positions (or about 10^{65}). Of course, there are lots of possible games that go much longer than the "typical" game, so this estimate is likely quite low.* If we guess that a single position can be represented in 32 bytes (8 bytes for a bitmap showing which squares are occupied, 4 bits for each occupied square to specify which piece is there, a few bits for whose turn it is, the number of times the position has been reached, and "win for white," "win for black," "tie," or "not yet determined," and a very optimistic assumption that the links to all of the possible successor positions can be squeezed into the remaining space), then all we need is about 10^{56} 32-gigabyte disk drives to store the data. With only an estimated 10^{70} protons in the universe, that may be difficult.

It will take quite a few rotations of our galaxy to generate all of those positions, so you can take advantage of bigger disk drives as they become available. Of course, the step to analyze all of the positions will take a bit longer. In the meantime, you might want to use a less complete analysis for your chess program. break

---

* Patrick Henry Winston, in his book *Artificial Intelligence*, (Addison-Wesley, 1992) provides a casual estimate of 10^{120}.

The exponential growth of the problem's size makes that technique unworkable for chess, but it is tolerable for tic-tac-toe:

```
use tic_tac_toe;        # defined earlier in this chapter


# exhaustive analysis of tic-tac-toe
sub ttt_exhaustive {


    my $game = tic_tac_toe->new( );


    my $answer = ttt_analyze( $game );
    if ( $answer > 0 ) {
        print "Player 1 has a winning strategy\n" ;
    } elsif ( $answer < 0 ) {
        print "Player 2 has a winning strategy\n";
    } else {
        print "Draw\n";
    }
}


# $answer = ttt_analyze( $game )
#    Determine whether the other player has won.  If not,
#    try all possible moves (from $avail) for this player.
sub ttt_analyze {
    my $game = shift;


    unless ( defined $game->prepare_moves ) {
        # No moves possible.  Either the other player just won,
        # or else it is a draw.
        my $score = $game->evaluate;
        return -1 if $score < 0;
        return 0;
    }


    # Find result of all possible moves.
    my $best_score = -1;
```